

Lecture Topics

- revisit some of last lecture (ABA problem, lock algos.)
- a few lock algorithms (wrap-up)
- ownership transfer & queue examples

Administrivia

- [none]

- ticket lock: take a number, wait your turn
 - one cache line for the dispenser
 - one cache line for the number being served
 - to lock, pull a number from the dispenser (fetch-and-increment)
 - check the display; when you see your number, you have the lock
 - when you're done, increment the display to unlock
 - memory traffic effects
 - only one thread changes the display on handoff
 - but all threads immediately want to share the new display value

```

void lock_ticket (ticket_lock_t* lock)
{
    int32_t mine = Fetch_and_Increment (&lock->dispenser);
    while (mine != lock->display);
}

void unlock_ticket (ticket_lock_t* lock)
{
    // need compiler and memory barriers here!
    lock->display++;
}

int32_t lock_anderson (anderson_lock_t* lock)
{
    int32_t mine = (Fetch_and_Increment (&lock->dispenser) %
                    N_CONTENTENDERS);
    while (0 == lock->slot[mine]);
    return mine;
}

void unlock_anderson (anderson_lock_t* lock, int32_t mine)
{
    lock->slot[mine] = 0;
    // need compiler and memory barriers here!
    lock->slot[(mine + 1) % N_CONTENTENDERS] = 1;
}

```

- Anderson lock
 - choose maximum number of contenders (power of 2)
 - allocate one cache line per possible contender
 - plus one line to hold a pointer to the current owner
 - to lock, fetch-and-increment the pointer
 - low bits tell you your slot
 - when your slot holds the “token” (say, a 1), you own lock
 - to unlock, write 0 into your slot, and 1 into the next slot
 - memory traffic effects
 - only the next thread’s cache line invalidated on lock handoff
 - best you can do without cache injection support
 - drawbacks
 - bound on threads may not be small (1 cache line each!)
 - pull in & immediately examine two cache lines rather than one, so slow when not contended compared with all previous
- can also build queueing locks and reader-writer locks
- example: Linux reader/writer locks with fetch-and-add
 - determine max threads N
 - start lock value at N
 - reader subtracts 1
 - writer subtracts N
 - negative result means failed acquisition
 - add same value back
 - wait until acceptable value seen (like T&T&S)
 - try again
- note that waiting for an acceptable value precludes livelock scenarios in which multiple threads constantly keep the lock value below the level at which any of them can succeed

Ownership Transfer

- critical sections must be small
 - large critical sections increase contention
 - more likely to be a bottleneck
 - more likely to require extra work to add locks
- instead, many use ownership transfer models
 - only one thread can touch a particular object
 - possibly temporary
 - under lock, change object state to “private”
 - other threads steer clear for a while
 - possibly always (with thread varying)
 - when thread is done, pass to next thread
 - use queue abstraction
 - message-passing on shared memory
- queue models
 - one to one (single producer, single consumer)
 - this structure you can build with loads and stores!
 - still need memory + compiler barriers on many ISAs
 - often used, but can use lots of memory and be challenging with dynamic threads
 - many to one (multiple producer, single consumer)
 - need synchronization primitives
 - efficient in memory
 - any thread can find queue for known target thread
 - slower in theory
 - but polling many queues is not fast
 - so competitive in practice
 - also, most communication patterns have limited degree